CLAIMS

1. A method for making a lithographic printing plate from an original containing continuous tones comprising the steps of:
- screening said original to obtain screened data
- scan-wise exposing a lithographic printing plate precursor according to said screened data, said lithographic printing plate precursor having on a support a surface capable of being differentiated in ink accepting and ink repellant areas upon said scan-wise exposure and an optional development step and
- optionally developing a thus obtained scan-wise exposed lithographic printing plate precursor,
characterized in that said screening is a frequency modulation screening.

2. A method according to claim 1 wherein said frequency modulation screening proceeds according to the following steps:
- selecting an unprocessed image pixel according to a space filling deterministic fractal curve or a randomized space filling curve and processing said unprocessed image pixel as follows:
- determining from the tone value of said unprocessed image pixel a reproduction value to be used for recording said image pixel on a recording medium,
- calculating an error value on the basis of the difference between said tone value of said unprocessed image pixel and said reproduction value, said unprocessed image pixel thereby becoming a processed image pixel,
- adding said error value to the tone value of an unprocessed image pixel and replacing said tone value with the resulting sum or alternatively distributing said error value over two or more unprocessed image pixels by replacing the tone value of each of said unprocessed image pixels to which said error value will be distributed by the sum of the tone value of the unprocessed image pixel and part of said error,
- repeating the above steps until all image pixels are processed.

3. A method according to claim 2 wherein said original having continuous tones is subdivided in matrices of unprocessed image pixels and all of said image pixels within a matrix is processed before a subsequent matrix is processed.

4. A method according to claim 1 wherein said lithographic printing plate precursor contains a photosensitive layer.

5. A method according to claim 1 wherein said lithographic printing plate precursor contains a heat mode recording layer containing a substance capable of converting light into heat.

6. A method according to claim 1 wherein said lithographic printing plate precursor contains a silver halide emulsion layer and an image receiving layer containing physical development nuclei and wherein subsequent to said scan-wise exposure said lithographic printing plate is developed using an alkaline processing liquid in the presence of developing agent(s) and silver halide solvent(s).

7. A method according to claim 1 wherein said scan-wise exposure is carried using a laser or LED.

add E'

Add G'

H2

Annex 1

```
typedef struct {
 int i;
 int j;
 } Index;

typedef struct {
 Index pt1;
 Index pt2;
 Index pt3;
 Index pt4;
 } Hilbert_Elem;

store_hilbert_elem(it,p)
 Itile *it;
 Hilbert_Elem *p;
{
static int n=0;
it->elem[n][0] = p->pt1.i; it->elem[n][1] = p->pt1.j; n++;
it->elem[n][0] = p->pt2.i; it->elem[n][1] = p->pt2.j; n++;
it->elem[n][0] = p->pt3.i; it->elem[n][1] = p->pt3.j; n++;
it->elem[n][0] = p->pt4.i; it->elem[n][1] = p->pt4.j; n++;
}

/*** Initiation for Recursive Calculation of Hilbert Scan ***/
hilbert_initiation(size,p)
 int size; Hilbert_Elem *p;
{
p->pt1.i = 1*size/4; p->pt1.j = 1*size/4;
p->pt2.i = 1*size/4; p->pt2.j = 3*size/4;
p->pt3.i = 3*size/4; p->pt3.j = 3*size/4;
p->pt4.i = 3*size/4; p->pt4.j = 1*size/4;
}
```

```
/*** Recursive Module to Calculate Hilbert Scan ***/
hilbert_propagation(it,p)
 Itile *it;
 Hilbert_Elem *p;
{
int i1,j1,i2,j2,i4,j4,l,li,lj,si,sj,orientation,length;
Hilbert_Elem p1,p2,p3,p4;

i1 = p->pt1.i;
j1 = p->pt1.j;
i2 = p->pt2.i;
j2 = p->pt2.j;
i4 = p->pt4.i;
j4 = p->pt4.j;

li = (i4 - i1)/2.0;
lj = (j4 - j1)/2.0;

orientation = (i4-i1)*(j2-j1) - (j4-j1)*(i2-i1);
l = (int)sqrt((double) li*li + lj*lj);

if(orientation < 0)
 {
 p1.pt1.i = p->pt1.i;            p1.pt1.j = p->pt1.j;
 p1.pt1.i -= (li+lj)/2.0;        p1.pt1.j -= (li+lj)/2.0;
 p1.pt2.i = p1.pt1.i+li;         p1.pt2.j = p1.pt1.j+lj;
 p1.pt3.i = p1.pt2.i+lj;         p1.pt3.j = p1.pt2.j-li;
 p1.pt4.i = p1.pt3.i-li;         p1.pt4.j = p1.pt3.j-lj;

 p2.pt1.i = p1.pt4.i+lj;         p2.pt1.j = p1.pt4.j-li;
 p2.pt2.i = p2.pt1.i+lj;         p2.pt2.j = p2.pt1.j-li;
 p2.pt3.i = p2.pt2.i+li;         p2.pt3.j = p2.pt2.j+lj;
 p2.pt4.i = p2.pt3.i-lj;         p2.pt4.j = p2.pt3.j+li;

 p3.pt1.i = p2.pt4.i+li;         p3.pt1.j = p2.pt4.j+lj;
 p3.pt2.i = p3.pt1.i+lj;         p3.pt2.j = p3.pt1.j-li;
 p3.pt3.i = p3.pt2.i+li;         p3.pt3.j = p3.pt2.j+lj;
 p3.pt4.i = p3.pt3.i-lj;         p3.pt4.j = p3.pt3.j+li;

 p4.pt1.i = p3.pt4.i-lj;         p4.pt1.j = p3.pt4.j+li;
 p4.pt2.i = p4.pt1.i-li;         p4.pt2.j = p4.pt1.j-lj;
```

```
 p4.pt3.i = p4.pt2.i-lj;          p4.pt3.j = p4.pt2.j+li;
 p4.pt4.i = p4.pt3.i+li;          p4.pt4.j = p4.pt3.j+lj;
 }
else
 {
 p1.pt1.i = p->pt1.i;             p1.pt1.j = p->pt1.j;
 p1.pt1.i -= (li+lj)/2.0;         p1.pt1.j -= (li+lj)/2.0;
 p1.pt2.i = p1.pt1.i+li;          p1.pt2.j = p1.pt1.j+lj;
 p1.pt3.i = p1.pt2.i-lj;          p1.pt3.j = p1.pt2.j+li;
 p1.pt4.i = p1.pt3.i-li;          p1.pt4.j = p1.pt3.j-lj;

 p2.pt1.i = p1.pt4.i-lj;          p2.pt1.j = p1.pt4.j+li;
 p2.pt2.i = p2.pt1.i-lj;          p2.pt2.j = p2.pt1.j+li;
 p2.pt3.i = p2.pt2.i+li;          p2.pt3.j = p2.pt2.j+lj;
 p2.pt4.i = p2.pt3.i+lj;          p2.pt4.j = p2.pt3.j-li;

 p3.pt1.i = p2.pt4.i+li;          p3.pt1.j = p2.pt4.j+lj;
 p3.pt2.i = p3.pt1.i-lj;          p3.pt2.j = p3.pt1.j+li;
 p3.pt3.i = p3.pt2.i+li;          p3.pt3.j = p3.pt2.j+lj;
 p3.pt4.i = p3.pt3.i+lj;          p3.pt4.j = p3.pt3.j-li;

 p4.pt1.i = p3.pt4.i+lj;          p4.pt1.j = p3.pt4.j-li;
 p4.pt2.i = p4.pt1.i-li;          p4.pt2.j = p4.pt1.j-lj;
 p4.pt3.i = p4.pt2.i+lj;          p4.pt3.j = p4.pt2.j-li;
 p4.pt4.i = p4.pt3.i+li;          p4.pt4.j = p4.pt3.j+lj;
 }

if(l > 1.0)
 { hilbert_propagation(it,&p1);
  hilbert_propagation(it,&p2);
  hilbert_propagation(it,&p3);
  hilbert_propagation(it,&p4); }
else /* termination */
 { store_hilbert_elem(it,&p1);
   store_hilbert_elem(it,&p2);
   store_hilbert_elem(it,&p3);
   store_hilbert_elem(it,&p4);
 return; }
}
```

```
main()
{
char name_path[32];
int size;
FILE *fp;
Itile it;

Hilbert_Elem p;
printf("enter name path under which the Hilbert path will be stored:
");
scanf("%s",name_path);
fp = fopen(name_path,"w");
printf("enter size of square path (in pixels, must be power of 2!!):
");
scanf("%d",&size);
size = 32;
alloc_itile(size*size,2,&it);
strcpy(it.descr,"hilbert_curve");
it.nr = size*size;
it.nc = 2;
it.min = 0;
it.max = size;
hilbert_initiation(size,&p);
hilbert_propagation(&it,&p);
write_itile(fp,&it);
}
```

Annex 2

```
permut_2D(seed,n,a)
 int seed,n,**a;
{
int i,*b,c[2],d[2];
b = (int *) ivector(n*n);
ran_perturb(seed,n*n,b);
/* replaces the n x n elements in vector b by a random permutation*/
c[0]=c[1]=n;
for(i=0;i<n*n;i++)
 {
 calc_index_from_lin_addr(2,c,b[i],d);
/* transforms the linear address b[i] into a coordinate pair in
vector d */
 a[d[0]][d[1]] = i;
 }
free_ivector(b);
}

/*** RECURSIVE CALCULATION OF 2D ORDER ***/
recurs_order_calc(sd,lv,tp,nb,ib,jb,od)
 int *sd,lv,*tp,nb,ib,jb;
 Itile *od;
{
int i,j,**ma,sz,ba;
sz = tp[lv];
ma = (int **) imatrix(tp[lv],tp[lv]);
permut_2D(sd,sz,ma);
for(i=0,ba=1;i<lv;i++)
 ba *= tp[i];
if(lv == 0)
 {
 for(i=0;i<sz;i++)
  for(j=0;j<sz;j++)
   {
   od->elem[nb+ma[i][j]*ba*ba][0] = ib+i*ba;
   od->elem[nb+ma[i][j]*ba*ba][1] = jb+j*ba;
   }
 return;
 }
for(i=0;i<sz;i++)
```

```
 for(j=0;j<sz;j++)
recurs_order_calc(sd,lv-1,tp,nb+ma[i][j]*ba*ba,ib+i*ba,jb+j*ba,od);
free_imatrix(sz,ma);
}

main()
{
char name_path[32];
int n,seed,level,topol[5],**order;
int size;
FILE *fp;
Itile it;

printf("enter name of path: ");
scanf("%s",name_path);
fp = fopen(name_path,"w");
/* "size" is the size of matrix over which error propagation will
take place */
size = 32;
/* this matrix will be "level" times recursively subdivided into
subsquares */
level = 4;
/* "topol" describes the subsequent size of these submatrices */
topol[0]=2; topol[1]=2; topol[2]=2; topol[3]=2; topol[4]=2;

alloc_itile(size*size,2,&it);
strcpy(it.descr,"cr_path");
it.nr = size*size;
it.nc = 2;
it.min = 0;
it.max = size;
seed = -1;
recurs_order_calc(&seed,level,topol,0,0,0,&it);
write_itile(fp,&it);
}
```